07011 – Abstracts Collection
Runtime Verification
— Dagstuhl Seminar —

Bernd Finkbeiner[1], Klaus Havelund[2], Grigore Roşu[3] and Oleg Sokolsky[4]

[1] Saarland University, Reactive Systems Group
Saarbrücken, Germany
finkbeiner@cs.uni-sb.de
[2] NASA's Jet Propulsion Laboratory, Laboratory for Reliable Software
Pasadena, California, USA
Klaus.Havelund@jpl.nasa.gov
[3] University of Illinois at Urbana-Champaign, Formal Systems Laboratory
Urbana, Illinois, USA
grosu@cs.uiuc.edu
[4] University of Pennsylvania, Department of Computer and Information Science
Philadelphia, Pennsylvania, USA
sokolsky@saul.cis.upenn.edu

Abstract. From January 2–6 2007 the Dagstuhl Seminar 07011 *'Runtime Verification'* was held in the International Conference and Research Center (IBFI), Schloss Dagstuhl. During the seminar, several participants presented their current research, and ongoing work and open problems were discussed. Abstracts of the presentations given during the seminar have been put together in this paper. The first section is an executive summary that describes the seminar topics in general.

Keywords. Program monitoring, dynamic program analysis, specification languages and logics, concurrency errors, program instrumentation, aspect-oriented programming, test oracles, fault protection, dynamic specification learning, combining static and dynamic analysis.

07011 Executive Summary – Runtime Verification

The 2007 Dagstuhl Seminar 07011 on *Runtime Verification*[1] was held from Tuesday January 2 to Saturday January 6, 2007. Over the past few years, runtime verification has emerged as a focused subject in program analysis that bridges the gap between the complexity-haunted field of fully formal verification methods and the ad-hoc field

[1] The website for the seminar:
http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=07011.

of testing. Other terms for this subject are: program monitoring, dynamic program analysis, and runtime analysis. Thirty researchers participated in the seminar and discussed their recent work and recent trends in runtime verification.

Joint work of: Finkbeiner, Bernd; Havelund, Klaus; Roşu, Grigore; Sokolsky, Oleg

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2008/1369

Combining Unit Test Coverage with Fault Injection

Cyrille Artho (National Institute of Informatics - Tokyo, J)

Testing application behavior in the presence of I/O failures is extremely difficult. The resources used for testing usually work without failure. Failures are usually not tested sufficiently. The Enforcer tool identifies such potential failures and automatically tests all relevant outcomes of such actions. It combines the structure of unit tests, coverage information, and fault injection. By taking advantage of a unit test infrastructure, performance can be improved by orders of magnitude compared to previous approaches.

Keywords: Software testing, fault injection, run-time verification

On the Design and Semantics of Trace Monitoring Features

Pavel Avgustinov (Oxford University, GB)

A lot of recent research has focussed on trace monitoring — the technique of observing program execution and triggering extra code when certain conditions are met. This is particularly interesting from the point of view of Runtime Verification, since many error conditions can be identified by program traces containing them, and thus a suitable monitor could perform the necessary instrumentation.

Tracematches are introduced as a particular example of a trace monitoring feature. Their design is presented, discussing the different choices made. A formal semantics for matching a tracematch to a program execution trace is presented; this is then refined into an operational semantics that can guide an implementation. A brief

comparison of the tracematch formalism to a "skipping semantics" shows skipping adds no additional expressiveness. We conclude by speculating on the possibility of a generic trace monitoring back-end.

Keywords: Trace monitoring, tracematches, runtime verification, semantics

Towards a Logical Framework for Tightly Coupled Monitoring and Evolution of Software Components

Howard Barringer (Manchester University, GB)

We outline a logical modelling approach to describe evolvable component systems. Here, an evolvable component system is built hierarchically from components, and each component at each level of the system may have an associated evolver process. The evolver's purpose is to monitor and possibly change the associated component, where evolutionary change may be determined purely internally from observations made by the evolver, or may be stimulated from the outside. We model such systems in a revision-based first-order logical framework in which the theory for the evolver is at a meta-level to the theory for the component. This enables evolutionary change (i.e. theory change) of the component to be induced by a state revision of the evolver at the meta-level.

Keywords: Evolvable Software, Run-time Monitoring, Logical Modelling, Revision Theory

Runtime Reflection in a Nutshell

Andreas Bauer (TU München, D)

Reactive distributed systems have pervaded everyday life and objects, but often lack measures to ensure adequate behaviour in the presence of unforeseen events or even errors at runtime. As interactions and dependencies within distributed systems increase, the problem of detecting failures which depend on the exact situation and environment conditions they occur in grows. As a result, not only the detection of failures is increasingly difficult, but also the

differentiation between the symptoms of a fault, and the actual fault itself, i. e., the cause of a problem.

We present a modular approach for analysing reactive distributed systems at runtime, in that we provide a framework, the runtime reflection framework, for detecting failures as well as identifying their causes. Our approach is based upon different layers of analysis, each of which is performed at runtime. The first layer performs monitoring of observable system behaviour, and comparing it with a reference behaviour often specified in terms of an LTL property. Layer 2 then tries to infer root causes for a detected discrepancy by performing a logic-based diagnosis on the results of the monitors. Finally, if the explanations are sufficiently detailed, a dedicated reconfiguration can be triggered in layer 3 of the framework to put the system back into a well-defined state, or to safely shut down the system.

Keywords: Runtime verification, LTL, diagnosis, reconfiguration

A Staged Static Program Analysis to Improve the Performance of Runtime Monitoring

Eric Bodden (McGill University - Montreal, CA)

In runtime monitoring a programmer specifies a piece of code to execute whe a trace of events occurs during program execution. Our work is based on tracematches, an extension to AspectJ, which allows programmers to specify traces via regular expressions with free variables. In this paper we present a staged static analysis which speeds up trace matching by reducing the required runtime instrumentation.

The first stage is a simple analysis that rules out entire tracematches, just based on the names of symbols. In the second stage, a points-to analysis is used, along with a flow-insensitive analysis that eliminates instrumentation points with inconsistent variable bindings. In the third stage the points-to analysis is combined with a flow-sensitive analysis that also takes into consideration the order in which the symbols may execute.

To examine the effectiveness of each stage, we experimented with a set of nine tracematches applied to the DaCapo benchmark suite. We found that about 25% of the tracematch/benchmark combinations had instrumentation overheads greater than 10%. In these cases

the first two stages work well for certain classes of tracematches, often leading to significant performance improvements. Somewhat surprisingly, we found the third, flow-sensitive, stage did not add any improvements.

Keywords: Aspect-oriented programming, runtime monitoring, static dataflow analysis

Aspect Mining ... And Then?

Silvia Breu (Cambridge University, GB)

As a program evolves it is easy to overlook that certain functionality is not or no longer properly encapsulated but scattered over many methods. Aspect mining identifies such cross-cutting concerns in a program to help migrating it to an aspect-oriented design or to simply understand a software system (better). However, the results retrieved from applying different aspect mining techniques can also be useful for runtime verification.

We will present two approaches and then envision how the results can be used for runtime verification: DynAMiT (Dynamic Aspect Mining Tool) analyses program traces reflecting the runtime behaviour of a system in search for recurring execution patterns of method relations. HAM (History-based Aspect Mining) applies formal concept analysis to a program's development history: method calls added across many locations within transactions are likely to be cross-cutting.

The identified cross-cutting concerns (aspect candidates) can be seen as rules with which, e.g., program execution traces have to comply, or which can help to categorise program runs into good and bad (faulty) runs. These rules can also be seen as some kind of specification which future changes to the system have still to fulfil, without requiring the programmer to actually write down the specification rules herself.

Keywords: Aspect mining, execution traces, formal concept analysis, mining software repositories, rules

An Aspect Oriented Runtime Verification System for C

Klaus Havelund (Jet Propulsion Laboratory, USA)

We present a framework, named RMOR, for monitoring the execution of C programs against a special variant of state machines that have liveness states as well as safety states. A liveness state has to be left eventually once entered, which is not the case for safety states. In a finite trace context this means before the end of a monitored trace. The state machines are written in a lexical (non-graphical) format in separate monitor files, similar to the manner in which aspects are normally written separate from the program they apply to. The state machine language has been inspired by the graphical state machine language RCAT developed at NASA's Jet Propulsion Laboratory, USA (Margaret Smith, 2006). Transitions between states are labeled with abstract event names and Boolean expressions over such. The abstract events are connected to code fragments in the monitored program using a pointcut language inspired by aspect oriented programming, specifically AspectJ. The system is implemented in the C instrumentation and analysis package CIL (and programmed in Ocaml), which turns out to be a convenient framework for the analysis and transformation of C programs needed here. The work can be extended to a full aspect oriented framework for C combined with runtime verification.

Keywords: Runtime verification, state machines, aspect oriented programming, C, CIL

Assertion-based Repair of Structurally Complex Data

Sarfraz Khurshid (Univ. of Texas at Austin, USA)

Programmers have long used assertions to characterize properties of code. An assertion violation signals a corruption in the program state. At such a state, it is standard to terminate the program, debug it if possible, and re-execute it. We propose a new view: instead of terminating the program, use the violated assertion as a basis of repairing the state of the program and let it continue.

We present a novel algorithm to repair structurally complex data. Given a structure that violates an assertion that represents its integrity constraints, our algorithm performs a systematic search based on symbolic execution to repair the structure, i.e., mutate it such that the resulting structure satisfies the constraints and is similar to the original one. Heuristics to prune search and minimize mutations enable efficient and effective repair.

Experiments using libraries and applications, such as a naming architecture, a database engine, and a file system, show that our prototype efficiently repairs complex structures—even those with thousands of objects—while enabling systems to recover from potentially crippling errors.

Keywords: Data structure repair, error recovery, runtime verification

Timed Games and Generation of Testing Strategies for Real-Time Systems Using UPPAAL Tiga.

Kim Gulstrand Larsen (Aalborg University, DK)

The talk will present the notion of timed game automata and the extension of the real-time verification tool UPPAAL for synthesizing winning strategies for such games with respect to both safety and liveness properties. It will be demonstrated how the tool can be used for generating test strategies from timed I/O automata specification with uncontrollable output and timing uncertainties.

Monitoring Real-Time Properties

Martin Leucker (TU München, D)

This paper presents a construction for runtime monitors that check real-time properties expressed in timed LTL (TLTL). Due to D'Souza's results, TLTL can be considered a natural extension of LTL towards real-time.

Moreover, a typical obstacle in runtime verification is solved both for untimed and timed formulae, in that standard models of linear temporal logic are infinite traces, whereas in runtime verification only finite system behaviours are at hand.

Therefore, a 3-valued semantics (*true, false, inconclusive*) for LTL and TLTL on finite traces is defined that resembles the infinite trace semantics in a suitable and intuitive manner. Then, the paper describes how to construct, given a LTL/TLTL formula, a deterministic monitor with three output symbols that reads a finite trace and yields its according 3-valued LTL/TLTL semantics. Notably, the monitor rejects a trace as early as possible, in that any minimal bad prefix results in *false* as a return value.

Keywords: Runtime Verification, LTL on finite traces, Realtime LTL

Joint work of: Bauer, Andreas; Leucker, Martin; Schallhart, Christian

Full Paper:
http://www4.in.tum.de/ leucker/Documents/Leucker/fsttcs06.pdf.gz

See also: Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'06), volume 4337 of Lecture Notes in Computer Science, Kolkata, India, December 2006. Springer-Verlag.

Online Testing, Monitoring, and Emulation of Real-Time Systems

Brian Nielsen (Aalborg University, DK)

We present a framework for online model-based testing for real-time systems. Given a timed automata model of the system under test and its assumed environment our techniques generates and executes tests online and in real-time against the real system under test. A key-point of the framework is explicit and separate description of the environment assumptions. This view is supported from theory to tool to applications.

We show how the explicit handling of environment assumptions enables us to use the testing tool for monitoring (online as well as offline) and for emulating real-time systems through a simple and elegant reconfiguration of the tool.

Monitoring Partial Order Snapshots

Doron A. Peled (Bar-Ilan Univ. - Ramat-Gan, IL)

When monitoring a concurrent executing, events are registered according to some arbitrary order. Still, one may want to check the existence of snapshots (global states) according to some different, yet equivalent, ordering. In this paper we show an algorithm for doing it. We extend the temporal logic LTL to reason about such snapshots and present a model checking algorithm.

Keywords: LTL, monitoring concurrency, partial order execution

Lola: Runtime Monitoring of Synchronous Languages

Sriram Sankaranarayanan (NEC - Princeton, USA)

We present a specification language and algorithms for the online and offline monitoring of synchronous systems including circuits and embedded systems. Such monitoring is useful not only for testing, but also under actual deployment. The specification language is simple and expressive; it can describe both correctness/failure assertions along with interesting statistical measures that are useful for system profiling and coverage analysis. The algorithm for online monitoring of queries in this language follows a partial evaluation strategy: it incrementally constructs output streams from input streams, while maintaining a store of partially evaluated expressions for forward references.

We identify a class of specifications, characterized syntactically, for which the algorithm's memory requirement is independent of the length of the input streams. Being able to bound memory requirements is especially important in online monitoring of large input streams. We extend the concepts used in the online algorithm to construct an efficient offline monitoring algorithm for large traces.

We have implemented our algorithm and applied it to two industrial systems, the PCI bus protocol and a memory controller. The results demonstrate that our algorithms are practical and that our specification language is sufficiently expressive to handle specifications of interest to industry.

Keywords: Runtime Monitoring; Temporal Logics; Synchronous Languages

Joint work of: Sankaranarayanan, Sriram; D'Angelo, Ben; Sipma, Henny; Finkbeiner, Bernd; Sanchez, Cesar; Manna, Zohar

Deadlock Avoidance in Distributed Systems

Henny Sipma (Stanford University, USA)

Deadlocks are a serious problem in concurrent systems. In centralized systems the three common methods to deal with deadlock are: prevention, detection, and avoidance. With deadlock prevention, absence of deadlock is guaranteed statically, usually at the price of severely restricting concurrency. With deadlock detection, it is assumed that deadlocks can be detected and undone at runtime by, for example, roll-back of transactions. This approach is common in databases. Deadlock avoidance methods take a middle route: at runtime requests for resources are processed by a protocol that checks whether granting the resource is safe, that is, whether it will not lead to a deadlock, by examining the resource state of all existing processes and the needs of the new process.

In distributed real-time and embedded (DRE) systems, deadlock avoidance is usually the only option. The price of prevention is too high in terms of performance lost by unnecessarily restricting concurrency. Detection is generally not possible in real-time systems, because of potentially unbounded delays, and roll-back is problematic for systems interacting with physical devices. Deadlock avoidance methods developed for centralized systems, however, are not applicable to distributed systems, because they rely on having access to a global resource state, which is impractical in distributed systems, because of the high communication overhead involved.

We have developed a deadlock avoidance approach for DRE systems that does not require any communication between components. It is applicable to systems in which resources are allocated in a nested manner. A concrete application is thread allocation in DRE systems in which processes make two-way method calls to other components, including nested upcalls, using a WaitOnConnection policy. We present protocols that use static information about the

global call graphs of the processes combined with runtime information about the local resource state to decide whether granting a resource is safe. We start with a basic protocol that provides absence of deadlock but does not guarantee liveness for individual processes, followed by more sophisticated protocols that guarantee liveness and provide distributed priority inheritance.

Joint work of: Sanchez, Cesar; Manna, Zohar; Gill, Chris; Subramonian, Venkita

Keywords: Deadlock avoidance, distributed systems, embedded systems

Monitoring Wireless Sensor Applications: Lessons from a Case Study

Oleg Sokolsky (University of Pennsylvania, USA)

We present a case study that considers the application of runtime verification technology to a wireless sensor application. The case study is performed using the SURGE TinyOS application for multihop routing, which executes on the Avrora TinyOS simulator. We discuss the problems we have encountered in the course of case study. The problems include unclear correctness properties for wireless network applications (indicating ad hoc development process) and inadequate tool support.

A wireless sensor network usually comprises of a collection of tiny devices with built-in processors that can gather physical and environment information such as temperature, light, sound, etc., and communicate with one another over radio. Many wireless sensor network applications sit on top of an operating system called TinyOS and are mostly written in nesC, an extension of C that provides a component-based programming paradigm. Most of wireless sensor network applications are developed and tested on a simulator before they are deployed in the environment because testing and debugging directly on physical devices are very difficult, especially when the network consists of many nodes, and may not provide enough information for debugging. A simulator usually produces detailed execution information and can help find errors. However, even with the simulator and nesC, the current state of development tools for wireless sensor network still requires very low-level programming, which

makes it hard for the developers to maintain a high-level view of the system operation.

During the validation stage, lack of sophisticated debugging tools for sensor networks makes it difficult to make the connection between a high-level functional or performance requirement and a particular aspect of system implementation.

This paper investigates a high-level approach to examine execution data from a simulator and analyze it using runtime verification. The technique 1) identifies and formally specifies high-level requirements for the system under development, 2) monitors a distributed wireless sensor network application using data provided by the simulator, and 3) checks for timing and dynamic properties to gain understanding of the relevant behaviors of wireless sensor nodes and to provide a systematic approach in finding bugs and errors. A particular runtime verification used in this paper is MaC. MaC provides specification languages capable of expressing functional, timing, and probabilistic properties to specify requirements or patterns of errors. Properties can, for example, examine periodic behaviors or identify a faulty node.

MaC then monitors and checks a wireless sensor network application against its specification by observing data produced by a simulator.

The motivation for applying the monitoring and checking technique to check wireless sensor network applications is threefold: 1) raise the development level for wireless sensor network, 2) provide a mechanism for understanding high-level behaviors of the system in terms of low-level observation, and 3) provide a tool based on the acceptance of the state of the art development tool for sensor networks.

Keywords: Runtime verification, wireless sensor network, Avrora simulator

Joint work of: Sammapun, Usa; Regehr, John; Lee, Insup; Sokolsky, Oleg

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2008/1371

Goldilocks: A Race and Transaction-Aware Runtime for Java

Serdar Tasiran (Koc University, TR)

Data races often result in unexpected and erroneous behavior. In addition to causing data corruption and possibly program crash, the presence of data races complicates the semantics of an execution which might no longer be sequentially consistent. Motivated by these observations, we have designed and implemented a Java runtime system that monitors program executions and throws a DataRaceException when a data race is about to occur. Analogous to other runtime exceptions, the DataRaceException provides two key benefits. First, accesses causing race conditions are interrupted and handled before they cause errors that may be difficult to diagnose later. Second, if no DataRaceException is thrown in an execution, it is guaranteed to be sequentially consistent. This strong guarantee helps to rule out many concurrency-related possibilities as the cause of erroneous behavior.

When a DataRaceException is caught, the operation, thread, or program causing it can be terminated gracefully. Alternatively, the DataRaceException can serve as a conflict-detection mechanism in optimistic uses of concurrency.

We start with the definition of data-race-free executions in the Java memory model. We generalize this definition to executions that use transactions, in addition to locks and volatile variables, for synchronization. We present a precise and efficient algorithm for dynamically verifying that an execution is free of data races. This algorithm generalizes the Goldilocks algorithm for data-race detection by handling transactions and providing the ability to distinguish between read and write accesses. We have implemented our algorithm and the DataRaceException in the Kaffe Java Virtual Machine. We have evaluated our system on a variety of publicly available Java benchmarks and a few microbenchmarks that combine lock-based and transaction-based synchronization. Our experiments indicate that our implementation has reasonable overhead. Therefore, we believe that the DataRaceException is a viable mechanism to enforce the safety of executions of multithreaded Java programs.

Joint work of: Tasiran, Serdar; Elmas, Tayfun; Qadeer, Shaz

Monitoring with Lower Runtime Overheads

Julian Tibble (Oxford University, GB)

A trace monitor observes the sequence of events in the execution of a program; when the sequence matches some specified temporal pattern, additional code is triggered. Runtime verification of temporal properties is the major application of such monitors.

One key feature of systems for trace monitoring is the ability to write temporal patterns containing variables, which range over objects in the observed program. This allows the programmer to write specifications about the intended behaviour of cliques of interacting objects.

Generating efficient instrumentation from patterns containing variables involves two major problems in designing the data structures that keep track of the matching state: avoiding memory leaks (that cause the observed program to use much more memory than it otherwise would), and allowing fast access to relevant state to prevent the instrumentation from making the observed program prohibitively slow. This talk demonstrates solutions to these problems that have been implemented in the tracematch system - part of the AspectBench Compiler.

Keywords: Trace monitoring, performance, optimisation

Pex, a Framework for Systematic Runtime Verification of .Net Programs

Nikolai Tillmann (Microsoft Research, USA)

Pex is a framework enabling runtime verification and white box testing of .Net programs. Pex generates test cases in a feedback loop; it runs a test case while monitoring its execution. Pex learns from recorded, detailed execution traces and derives additional test cases using a constraint solver. The result is a minimal test suite with maximal code coverage. Pex is integrated into Visual Studio and other unit test frameworks. Pex also contains a dynamic property checker, and a module to infer likely intended program behavior.

While only recently released internally within Microsoft, Pex has already found several interesting bugs in the shipped code.

Monitoring, Fault Diagnosis and Testing Real-time Systems using Analog and Digital Clocks

Stavros Tripakis (Cadence Labs - Berkeley, USA)

We give an overview of known methods for monitoring, fault diagnosis and testing problems for real-time systems using timed automata as the main model. We present techniques for constructing monitors/diagnosers/testers with analog or digital clocks. We list a number of open problems in the field.

Keywords: Monitoring, fault diagnosis, testing, timed automata

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2008/1370

Taming Interface Specifications

Lenore Zuck (Univ. of Illinois - Chicago, USA)

Software is often being assembled using third-party components where the developers have little knowledge of, and even less control over, the internals of the components comprising the overall system. One obstacle to composing agents is that current formal methods are mainly concerned with "closed" systems that are built from the ground up. Such systems are fully under the control of the user. Hence, problems arising from ill-specified components can be resolved by a close inspection of the systems. When composing systems using "off-the-shelf" components, this is often no longer the case.

The talk addresses the problem of *under-specification*, where an off-the-shelf component does only what it claims to do, however, it claims more behaviors than it actually has and that one wishes for, some of which may render it useles. Given such an under-specified module, we propose a method to automatically synthesize some safety properties from it, that would tame its "bad" behaviors. The advantage of restricting to safety properties is that they are monitorable. The safety properties are derived using an automata-theoretic approach. When restricting to omega-regular langauges, there is no maximal safety property. We construct an increasing sequence of safety properties. We also show how to construct an infinite-state automata that can capture any safety property that is contained in the original specifications.

Joint work of: Sistal, Prasad; Zuck, Lenore; Steffen, Bernhard; Margaria, Tiziana